An Improved Approach to Reinforcement Learning in Computer Go

Michael Dann
School of Computer Science and Information Technology
RMIT University
Melbourne, Victoria 3000
Email: Michael.Dann@rmit.edu.au

Fabio Zambetta
School of Computer Science and Information Technology
RMIT University
Email: Fabio.Zambetta@rmit.edu.au

John Thangarajah
School of Computer Science and Information Technology
RMIT University
Email: John.Thangarajah@rmit.edu.au

***Abstract*—Monte-Carlo Tree Search (MCTS) has revolutionised Computer Go, with programs based on the algorithm achieving a level of play that previously seemed decades away. However, since the technique involves constructing a search tree its performance tends to degrade in larger state spaces. Dyna-2 is a hybrid approach that attempts to overcome this shortcoming by combining Monte-Carlo methods with state abstraction. While not competitive with the strongest MCTS-based programs, the Dyna-2-based program *RLGO* achieved the highest ever rating by a traditional program on the 9×9 Computer Go Server. Plain Dyna-2 uses ϵ-greedy exploration and a flat learning rate, but we show that the performance of the algorithm can be significantly improved by making some relatively minor adjustments to this configuration. Our strongest modified program achieved an Elo rating 289 points higher than the original in head-to-head play, equivalent to an expected win rate of 84%.**

I. Introduction

The Dyna-2 algorithm [1, Chapter 7] is a hybrid approach to sequential decision-making that combines state abstraction with Monte-Carlo methods. Action-values are calculated as the sum of two components: a "long-term memory" trained on experience from real games and a "short-term memory" trained on simulated experience. Both memories are trained using Temporal Difference Learning [2, Chapter 6]. The short-term memory acts as a correction to the long-term memory. The intuition behind this is that a feature may be valuable most of the time but weak in certain situations, or vice-versa. To take an example from chess: bishops are generally valued slightly higher than knights, but in situations where the board is locked, the knight's ability to jump over squares may make it more useful than a bishop.

An agent based on Dyna-2, *RLGO*, showed significant promise in the challenging domain of Computer Go, surpassing the performance of all previous traditional (i.e. non-MCTS-based) programs on the 9×9 Computer Go Server [1, Chapter 7]. However, there was reason to believe that its performance could be improved. Its playing strength did not increase when certain additional features were added to its state representation that, from a human perspective, should have enabled better decision-making. The program's inability to make use of the new information suggested a problem with the learning algorithm.

While Dyna-2's performance in Go remains well short of the strongest MCTS-based programs, our interest in improving the algorithm is twofold: Firstly, Go is a convenient testbed for AI techniques, but real-world problems often have much larger state spaces and Dyna-2's use of state abstraction is designed to make it scale better than pure simulation approaches. Secondly, fast heuristics for Go (such as Dyna-2's action-value function) can be used to improve the random playouts in MCTS by biasing them toward more sensible moves. In fact, the first program to achieve master level in 9x9 Go, *MoGo* [3], used Dyna-2 for this purpose in its early development.

The contribution of this paper is to demonstrate that Dyna-2's performance in Computer Go can be significantly improved by making two relatively minor modifications:

- Replacing the algorithm's flat learning rate with a decaying rate allows feature values to be updated faster early on when they are most uncertain then fine-tuned later as learning progresses. The benefit of this modification to the long-term memory was negligible because the agent can simply be trained offline with a low, flat learning rate until it ceases to improve. However, the benefit to the short-term memory was significant because simulations must be performed online and therefore training efficiency is important.

- The original algorithm uses an ϵ-greedy policy during training, which means that exploratory moves are chosen at uniform random. We replaced ϵ-greedy with the softmax policy [2], which favours stronger-looking exploratory moves. This modification significantly improved the performance of both memories, since the training scenarios became more representative of competitive play. A particular finding of note was that ϵ-greedy suffered badly from a tradeoff between simulation quality and explorative depth, while softmax was far less compromised.

To quantify the skill difference between different versions of *RLGO* we used the Elo scale [4], which is a standard measure of performance in the Go literature [1], [5], [6], [7]. Ratings were calculated via *BayesElo* [8], using the publicly available Go program *GnuGo* [9] to anchor the scale, as is standard practice.

The combination of the modifications described above yielded an improvement of 289 Elo, equivalent to an expected win rate of 84% in head-to-head play against the original program. Having said this, the notion of a "true" Elo rating

is ill-posed since ratings are only relative to the player pool from which they were calculated. It is possible that the modified programs' edge would not be so large against a wider population of Go players.

The remainder of this paper is structured as follows:

In section II we briefly discuss the Monte-Carlo methods that revolutionised Computer Go, then provide a more detailed explanation of Dyna-2 and other important background concepts. In section III we describe our modifications to *RLGO* in depth. In section IV we conduct experiments to determine the best parameter values for the modified programs. Once so determined, we play tournaments between the original and modified programs in order to calculate and compare Elo ratings. Finally, in section V we summarise our findings and suggest ideas for further research.

II. Background and Related Work

In this section we provide a brief overview of the current state-of-the-art in Computer Go, namely the MCTS-based techniques that revolutionised the field. We then explain the hybrid approach of Dyna-2, how it was applied to Go and the concepts required to understand our modifications to it.

A. Current State-of-the-Art

The majority of the strongest Go programs today are based around UCT [10], a variant of MCTS. The algorithm incrementally builds a game tree according to a best-first search, evaluating each node by considering the average result of playouts simulated from the corresponding board state. The key to UCT's success was its treatment of the *exploration-exploitation dilemma* [2], i.e. whether one should continue exploring the most promising line of play or try to find other lines that might be even stronger. UCT calculates confidence intervals for the value of each move, then chooses the action with the highest upper confidence bound. In this sense it may be regarded as an optimistic policy. Since confidence intervals are widest when few simulations have been performed, UCT favours moves that have been less explored.

In its purest form, the actions chosen beyond the leaf node are picked at uniform random, but generally some form of heuristic knowledge is used to produce more realistic playouts. The strong programs *Crazy Stone* [11], [5], *MoGo* [3], [12] and *Fuego* [13] all take this approach. For example, Crazy Stone uses pattern knowledge obtained through supervised learning over human games to guide its random playouts [5].

A particularly well-known heuristic for Go is *rapid action-value estimation (RAVE)* [6], [14], an extension of the *all-moves-as-first (AMAF)* [15] idea. RAVE estimates the action-value of a move as the average result of playouts when that move was chosen at *any* later stage, not just on the next move. This heuristic can theoretically be applied in any environment where the sequence of actions is transposable, but it should be noted that its strength in Go arises largely from the nature of the game.

B. Dyna-2

The UCT algorithm revolutionised Computer Go, but its initial success was far greater on a reduced 9×9 board than it was for the full 19×19 game, which has a significantly larger state space and branching factor [16]. The algorithm's performance scales well with increasing hardware power [17], and it is feasible that this alone could see computers surpass humans in 19×19 Go, but there are many real life problems with far larger state spaces that will not be solved in this way.

A common way to cope with large state spaces is to use some form of *state abstraction* [1, Section 1.3.2]. Rather than evaluating states based on their full details, the value function is calculated in terms of some higher level *features*. For example, a feature in chess might be the existence of doubled pawns. Once the approximate value of doubled pawns is known, it can be used in the evaluation of all board states where doubled pawns occur.

The *Dyna-2* algorithm [1, Chapter 7] maintains two sets of features: a *long-term memory* trained on real experience and a *short-term memory* trained using simulated experience. The long-term memory learns broadly accurate feature values over many games, which may be thought of as rules-of-thumb, while the short-term memory learns a correction that applies only to the current state of the board. Both memories are trained using *logistic TD(λ)* [1, Appendix A.2] with ϵ-greedy policy improvement.

To apply Dyna-2 to a specific domain, it is important to choose a feature set that encodes useful information about the environment. *RLGO* uses a state representation called "local shape features" [7], which are defined as squares of size 1x1 up to 3x3, where squares containing different combinations of stones are considered to be distinct features (see Figure 1). Experienced Go players will realise that this is a naïve approach, since distant stones may exert a strong influence locally, but finding a state representation for Go that is both simple and effective has proven to be extremely difficult [18].[1] Local shape features are distinguished between *location-dependent features* (where the configuration of stones occurs at a specific board location) and *location-independent features* (where the configuration of stones appears anywhere on the board).

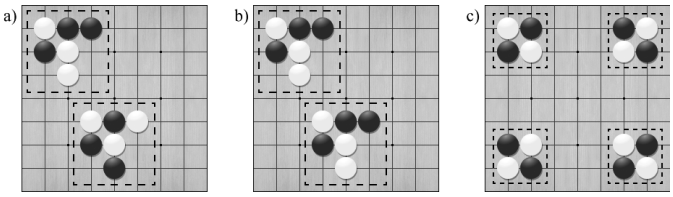

Fig. 1. a) Despite having the same shape, these 3x3 features are different because the stone colours do not match. b) These are different location-dependent features, but the same location-independent feature. c) Due to the rotational symmetries of the board, these features are considered to be identical in both location-dependent and location-independent terms.

The full Dyna-2 algorithm attempts to model non-deterministic environments, but since Go is a deterministic game we present a simplified version in Algorithm 1. Note that the combined action-value function $\hat{Q}$ is based on both the long- and short-term feature values (θ and $\bar{\theta}$), while Q is

[1] This may be about to change. In results yet to be published, a deep convolutional neural network was trained to learn a strong evaluation function for Go. [17]

based on the long-term memory only. In other words:

$$Q(s,a) = \phi(s,a) \cdot \theta \tag{1}$$

$$\hat{Q}(s,a) = \phi(s,a) \cdot \theta + \bar{\phi}(s,a) \cdot \bar{\theta} \tag{2}$$

where $\phi(s,a)$ is the set of long-term memory features present after executing a in state s and $\bar{\phi}(s,a)$ is the set of short-term memory features present. ϕ and $\bar{\phi}$ will be identical if the same feature set is used for both memories, but this need not be the case. Both memories explore using the combined action-value function, $\hat{Q}$, but the TD-error for the long-term memory is only calculated in terms of Q.

Algorithm 1. Dyna-2 (simplified for deterministic environments)

```
procedure LEARN
    θ ← 0                                   ▷ Reset long-term memory
    loop
        s ← s_0                             ▷ Start a new game
        θ̄ ← 0                               ▷ Reset short-term memory
        e ← 0                               ▷ Clear LT eligibility trace
        SIMULATE(s)
        a ← ε-greedy(s; Q̂)
        while s is not terminal do          ▷ Play until game over
            (s', r) ← execute(a)            ▷ Execute action
            SIMULATE(s')
            a' ← ε-greedy(s'; Q̂)
            δ ← r + Q(s', a') − Q(s, a)      ▷ LT TD-error
            θ ← θ + αδe                     ▷ Update LT memory
            e ← λe + φ                      ▷ Update LT eligibility trace
            s ← s', a ← a'
        end while
    end loop
end procedure

procedure SIMULATE(s)
    s̄ ← s                                   ▷ Copy state for simulating
    for i = 1 to simCount do
        ē ← 0                               ▷ Clear ST eligibility trace
        ā ← ε̄-greedy(s̄; Q̂)
        while s̄ is not terminal do          ▷ Simulate full playout
            (ŝ, r̄) ← execute(ā)             ▷ Execute action
            â ← ε̄-greedy(ŝ; Q̂)
            δ̄ ← r̄ + Q̂(ŝ, â) − Q̂(s̄, ā)      ▷ ST TD-error
            θ̄ ← θ̄ + ᾱδ̄ē                    ▷ Update ST memory
            ē ← λ̄ē + φ̄                      ▷ Update ST eligibility trace
            s̄ ← ŝ, ā ← â
        end while
    end for
end procedure
```

C. Softmax Exploration

Unlike the ϵ-greedy policy [2, Chapter 2.2], which explores at uniform random, the *softmax* policy is biased towards moves with higher action-values. The action selection distribution under softmax is governed by the following formula:

$$\Pr(\text{action } a \text{ selected}) = \frac{e^{Q_t(a)/\tau}}{\sum_{b \in A_t} e^{Q_t(b)/\tau}} \tag{3}$$

where A_t is the set of all possible actions at time t, τ is a parameter called the *temperature*, and $Q_t(a)$ is the action-value of move a at time t. The most promising move will be chosen with the greatest frequency, the second most promising move will be chosen with the second greatest frequency, and so on.

III. Modifications to Dyna-2

We now describe our modifications to the Dyna-2 algorithm. Section III-A covers the introduction of learning rate decay. Section III-B explains how we replaced the algorithm's ϵ-greedy policy with softmax.

A. Introducing Learning Rate Decay

The optimal learning rate curve for either memory could theoretically be a complicated function of training time and other factors, but for simplicity we opted for functions of the form

$$\alpha(n) = \frac{\alpha_0}{1 + \beta n} \tag{4}$$

where $\alpha(n)$ is the learning rate after n training games, a_0 is the initial learning rate and β controls how fast the learning rate decays. This choice was motivated by Monte-Carlo evaluation, where the learning rate is $1/n$. While probably not optimal, for our broader aim of determining whether learning rate decay might be a worthwhile modification to Dyna-2, we believed that this curve would suffice.

For the curve described by Equation 4, the average learning rate over N games is:

$$\alpha_{avg} = \frac{1}{N} \sum_{n=0}^{N-1} \frac{\alpha_0}{1 + \beta n} \tag{5}$$

which can be approximated by the integral

$$\frac{1}{N} \int_0^N \frac{\alpha_0}{1 + \beta n} dn \tag{6}$$

$$= \frac{\alpha_0}{\beta N} \log_e(1 + \beta N) \tag{7}$$

This formula allowed us to estimate the optimal decay curve from a series of data points. We ran the learning algorithm over a range of game counts and flat learning rates, then conducted a tournament between the resultant programs to determine their Elo ratings. For each training game count we observed the learning rate that gave rise to the strongest program, then used Equation 7 to find the values of α_0 and β that best fit our results.

B. Replacing ϵ-greedy with Softmax

The extent of exploration under the softmax policy is controlled by the temperature parameter, τ. The higher its value, the closer the policy becomes to uniform random. The lower it is, the greedier selection becomes. It is possible to estimate a sensible range for τ by experimenting with sample values in equation 3. For example, suppose that we are given a choice between two moves, a_1 and a_2, such that the action value of move a_1 is 0.1 greater than the value of a_2. Under a loose probabilistic interpretation of the action-value (which is

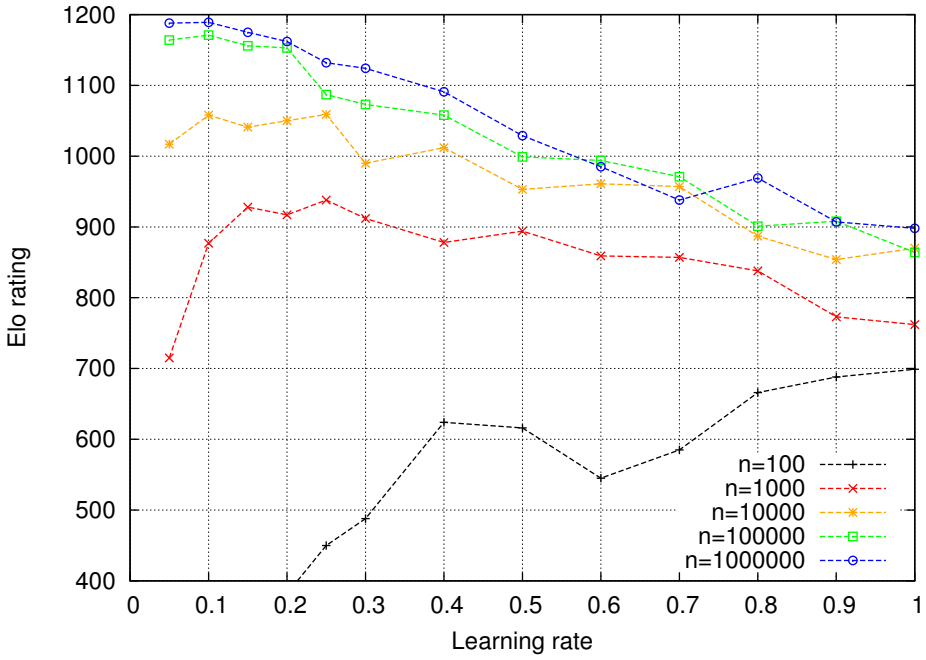

Fig. 2. Elo rating of the long-term memory vs. learning rate after averaging over ten training runs.

somewhat justified by the fact that Dyna-2 applies a logistic squashing function to its value function [19]), this means that a_1 is 10% more likely to lead to a win than a_2. From the softmax equation, the relative probability of a_1 being played compared to a_2 is

$$\frac{Pr(\text{move} = a_1)}{Pr(\text{move} = a_2)} = \frac{e^{Q_t(a_1)/\tau} * \sum_{b \in A_t} e^{Q_t(b)/\tau}}{e^{Q_t(a_2)/\tau} * \sum_{b \in A_t} e^{Q_t(b)/\tau}} = \frac{e^{(Q_t(a_2)+0.1)/\tau}}{e^{Q_t(a_2)/\tau}} = e^{0.1/\tau}$$

If $\tau = 0.01$ then a_1 will be chosen around 22,000 times more often than a_2, which is probably not explorative enough. On the other hand, if $\tau = 0.1$ then a_1 is only about 2.7 times more likely to be played than a_2, which intuitively may be too explorative given that a_2 reduces winning chances by 10%. Therefore, we expected the optimal value for τ to lie somewhere around the range of 0.01 – 0.1. From here we took a trial-and-error approach to more precisely estimate the optimal value of τ in each memory.

IV. Experimental Results

To determine appropriate values for the newly introduced parameters, we began by limiting the programs to the use of one memory only, first introducing learning rate decay (Section IV-A) then softmax exploration in parallel (Section IV-B). In Section IV-C we apply the combined modifications to the full Dyna-2 algorithm and determine the overall impact on playing strength.

A. Learning Rate Decay

Since the optimal decay curve could theoretically differ between the long- and short-term memories, we conducted separate experiments for each component.

1) Long-Term Memory: To determine an appropriate shape for the learning rate decay curve, we trained a group of programs over a range of flat learning rates and training game counts, then played a tournament between them to determine their Elo ratings. Since the strength of the resultant programs varied after different training runs we repeated the experiment ten times and took average ratings. Our results are summarised in Figure 2.

Unfortunately, close inspection of these results reveals that learning rate decay is unlikely to yield much improvement. For all training game counts other than $n = 100$, a flat learning rate of $\alpha = 0.1$ (which is *RLGO*'s default setting) is either optimal or very close to optimal. For large n, using $\alpha \leq 0.1$ allows slightly better convergence of feature values, but we can see that the difference in playing strength is minimal. Introducing a decay curve is likely to help for the first 100 games or so, but the long-term memory can easily be trained offline over millions of games so this advantage is hardly significant.

Nonetheless, we proceeded to fit a curve to the data and found that values of $\alpha_0 = 0.25$ and $\beta = 7 \times 10^{-6}$ in Equation 4 provided a satisfactory fit. We trained ten programs over 1,000,000 games using this decay curve and ten using the optimal flat rate, then played a tournament between them to see if the modified programs had any advantage. As expected, there was no significant difference between their performance,

with both program types rated ≈ 1150 Elo.

2) Short-Term Memory: In the short-term memory there was an additional consideration to be made with regard to the learning rate: while the long-term memory learns from scratch, the short-term memory retains its knowledge from one move to the next. Therefore there is a tradeoff between preserving useful information from the previous position and adjusting quickly when the character of the board changes (for example, when the queens are removed from the board in Chess).

To account for this, we tried a series of curves with different initial learning rates (α_0) but the same average rate over 10,000 simulations. We then played a tournament between the different versions, the results of which are summarised in Table I.

Rank	Program Name	α_0	β	Elo	+/-
1	decayA	0.2	0.00025	1954	37
2	decayB	1.0	0.0036	1908	36
3	decayC	0.8	0.0026	1896	35
4	baseline	0.2	–	1881	35
5	decayD	0.6	0.0017	1862	35
6	decayE	0.4	0.00093	1758	38

TABLE I. ELO RATINGS FOR SEVERAL VERSIONS OF THE SHORT-TERM MEMORY USING DIFFERENT DECAY CURVES. ELO ERROR BOUNDS REPRESENT 95% CONFIDENCE INTERVALS.

The best performing modified program, *decayA*, was rated 73 Elo higher than the baseline. Taking error bounds into account, its likelihood of superiority over the baseline was 99.8%.

Learning rate decay increases the efficiency of training, which is of greatest advantage when training time is limited. While the long-term memory could be trained offline indefinitely with a small learning rate, it is impractical to simulate for so long during a live game. With 10,000 simulations per move the above tournament took several days to complete on our hardware, while the long-term memory could easily be trained over millions of games. It is likely that the modified programs' advantage would diminish if more simulations were allowed per move.

On the other hand, learning rate decay in the short-term memory might prove more beneficial in real-world tournaments, since programs are normally bound by *time* per move, not some number of simulations. The modified algorithm is more robust to early termination because the average learning rate increases when the training period is truncated.

The relatively low initial learning rate of 0.2 for the best curve suggests that retaining information from the previous board state was more important than adjusting quickly to positional changes. However, the second best performing curve ($\alpha_0 = 1.0$, $\beta = 0.0036$) employed the opposite strategy, and those with middling values of α_0 actually performed far worse. Therefore it appears that it was better for the programs to focus on one priority rather than compromise on both.

B. Softmax Exploration

Since the optimal setting for softmax's temperature parameter, τ, could theoretically differ between the long- and short-term memories we conducted separate experiments for each component once again. For the modified programs we retained the best learning rate decay curves from the preceding sections. The baseline programs still use a flat rate.

1) Long-Term Memory: To optimise the temperature setting we conducted a tournament between several modified programs with different values of τ. For investigative purposes, we also recorded the percentage of occasions on which each program chose greedily. The results of this tournament are shown in Figure 3.

The ratings maxima occurs at $\tau = 0.07$, where the program chose greedily 53% of the time. We can conclude that lower values of τ provided insufficient variety during training, while larger values resulted in too many poor moves being made. When move selection is too random, the program effectively learns to specialise in unusual positions that are not representative of tournament play.

The default exploration rate for the original *RLGO* is 10% (i.e. 90% greedy selection), which is far greedier than our strongest softmax variant. The reason for this is that a softmax policy that chooses greedily 90% of the time is effectively far less "exploratory" than an ϵ-greedy policy that does likewise. Softmax tends to pick the second or third most promising option when it explores, whereas ϵ-greedy is equally likely to try all exploratory moves. The nature of Go is such that softmax is arguably more natural; there are often several strong lines of play, but a completely random move is likely to be bad.

To test this theory and measure the impact on playing strength we played a tournament between ten programs trained with softmax and ten trained with ϵ-greedy selection. The modified programs did in fact significantly outperform the baseline. Their average Elo rating was 119 points higher, which corresponds to an expected win rate of 66% in a head-to-head match. Out of the player pool of twenty programs, nine of the modified programs ranked inside the top eleven, while all of the bottom five programs were trained using ϵ-greedy.

2) Short-Term Memory: In his PhD thesis [1], Silver recognised that there is an inherent issue in using the short-term memory as the basis for the simulation policy. Since the short-term memory is designed to learn a *temporary* correction applicable to the *current* board state only, a policy based on the short-term memory is likely to become weaker the further each playout progresses. To account for this, Silver configured *RLGO* to switch to a handcrafted policy once a simulation exceeds six moves. The handcrafted policy was taken from a strong MCTS-based program, *Fuego* [13]. The policy consists of a hierarchy of common-sense rules, such trying to save stones that are placed in atari and cutting opponent stones when they threaten to connect. We retained this approach for our modified programs; softmax replaces ϵ-greedy for the first 6 simulated moves, then Fuego's default policy is used.

To determine the right level of exploration in the simulation phase we created a set of softmax programs over a range of τ values and played them against a set of baseline programs created over a range of ϵ. All programs were allowed 10,000 simulations per move. The results of this tournament are shown in Table II.

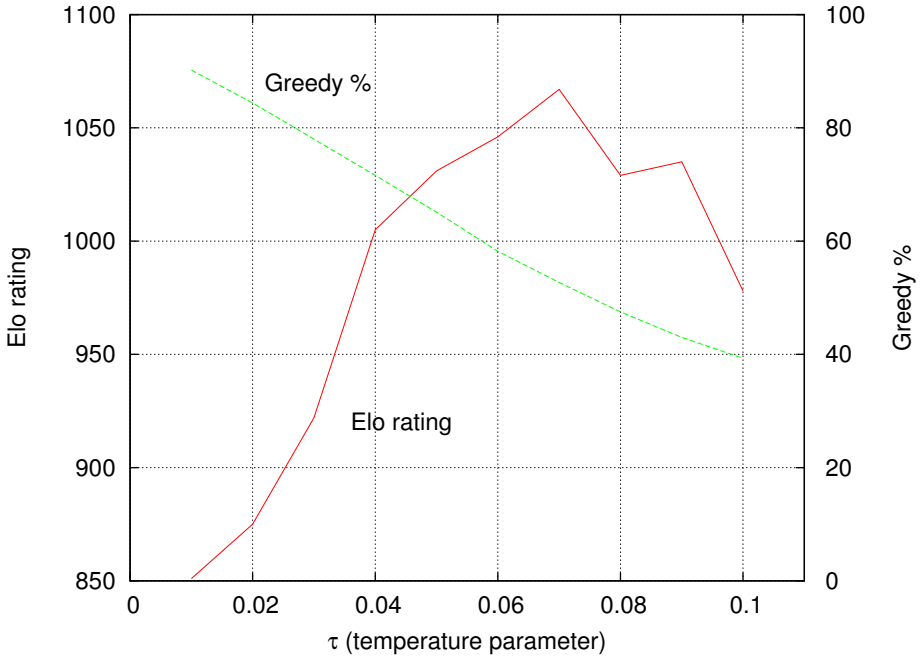

Fig. 3. Elo rating and greedy move percentage vs. τ in the long-term memory.

The best modified programs significantly outperformed the baseline, with an Elo difference of 223 between the strongest representatives (*softmax07* and *baseline04*). This corresponds to a 78% win rate for *softmax07* in head-to-head play.

Rank	Name	τ	ϵ	Elo	+/-
1	softmax07	0.07	–	1921	39
2	softmax08	0.08	–	1906	38
3	softmax06	0.06	–	1759	35
4	baseline04	–	0.4	1698	32
5	baseline06	–	0.6	1684	34
6	softmax05	0.05	–	1668	34
7	baseline03	–	0.3	1645	32
8	baseline05	–	0.5	1579	36
9	softmax04	0.04	–	1516	35
10	baseline07	–	0.7	1470	40

TABLE II. ELO RATINGS FOR THE SHORT-TERM MEMORY WHEN CONFIGURED TO USE DIFFERENT SIMULATION POLICES. ELO ERROR BOUNDS REPRESENT 95% CONFIDENCE INTERVALS.

Note that the strongest ϵ-greedy program used $\epsilon = 0.4$, which is far higher than the optimal setting of $\epsilon = 0.1$ in the long-term memory. ($\epsilon = 0.1$ performed so poorly in our preliminary experiments with the short-term memory that we excluded it from the above tournament.) The long-term memory can afford to explore relatively infrequently because its training time is essentially limitless, but the short-term memory must ideally analyse all critical lines before the simulation limit is exceeded. When $\epsilon = 0.1$, the program only tries a new move 10% of the time, and since it does so at uniform random its chances of finding an important move are very low.

The solution is naturally to increase the value of ϵ, but the downside of doing so is that it significantly degrades the quality of the simulations. The program wastes time analysing strange positions that arise from making many random moves in the immediate future. Softmax does not suffer nearly as badly from this issue, partly because it plays greedily less often, but also because its exploratory moves tend to be more reasonable.

C. Dyna-2 with Full Modifications

Having determined the best parameters for each memory individually, we were ready to apply the modifications to the full Dyna-2 algorithm. However we soon discovered an interaction effect that we had not counted on: softmax exploration became less explorative in the simulation phase once the long-term memory was enabled. The combined memories were actually weaker than the short-term memory alone!

We believe the reason for this behaviour was as follows: When the short-term memory is used in isolation its feature values get reset to zero at the start of every game. Therefore, the policy is close to uniform random early on, but as training progresses the policy's exploratory scope becomes narrower as it avoids moves with low action-values. Introducing the long-term memory means that the combined value function is biased from the start of the game, but since the long-term memory corresponds to a rather weak player, narrowing the scope of exploration based on its valuations is fraught with danger.

Our solution to this problem was to simply increase the

value of τ for the modified programs. Initial results with this approach were promising, so we proceeded to conduct the final tournament. We included several modified programs covering $0.07 \leq \tau \leq 0.13$ in 0.01 increments for the simulation phase. We also included baseline programs covering a range of ϵ, since we could not be certain that the optimal setting of $\epsilon = 0.4$ from Section IV-B2 would still be optimal against the new player pool. To confirm that the combined memories were now stronger than the short-term memory alone, we included the strongest programs from the simulation-only experiments (which we renamed *simOnlyB'line* and *simOnlyMod* here for clarity). For completeness, we also included a program using *RLGO*'s "tournament" simulation policy[2], which was created for competitive play. The results of this tournament are shown in Table III.

Name	τ	ϵ	α decay?	Elo	+/-
modified10	0.10	–	Yes	1927	36
modified13	0.13	–	Yes	1892	35
modified11	0.11	–	Yes	1878	34
modified08	0.08	–	Yes	1858	37
modified09	0.09	–	Yes	1853	37
simOnlyMod	0.07	–	Yes	1827	35
modified07	0.07	–	Yes	1785	35
modified12	0.12	–	Yes	1777	31
t'mentPol	–	see footnote	No	1760	33
baseline06	–	0.06	No	1638	36
baseline03	–	0.03	No	1632	35
baseline05	–	0.05	No	1558	38
simOnlyB'line	–	–	No	1511	41
baseline04	–	0.04	No	1480	44

TABLE III. THE RESULTS OF THE FINAL TOURNAMENT. THE "τ" AND "ϵ" COLUMNS REFER TO THE SHORT-TERM MEMORY. "α DECAY" REFERS TO BOTH MEMORIES WHERE APPLICABLE. ELO ERROR BOUNDS REPRESENT 95% CONFIDENCE INTERVALS.

The strongest modified program (*modified10*) crushed the strongest baseline program (*baseline06*) by 289 Elo, equivalent to an expected win rate of 84% in a head-to-head match. This appears to be an excellent result, but it needs to be taken with a grain of salt. Elo ratings are only relative to the player pool from which they were calculated and are not an absolute measure of player skill. We suspect that the modified programs play a similar style to the original *RLGO*, since they still view the board in terms of local shape features. It is possible that this saw them dominate the baseline despite a relatively small gap in skill, although further experiments against a wider player pool would be required to confirm this hypothesis.

V. CONCLUSIONS AND FURTHER WORK

We have shown that the performance of Dyna-2 in Computer Go can be significantly improved by introducing learning rate decay and by replacing ϵ-greedy selection with softmax. Aside from the headline result of 289 Elo gained overall, in the course of our experiments we revealed some interesting findings:

- Since the long-term memory can be trained offline over many games, the specific learning rate used is not particularly important, so long as it is low enough for feature values to converge eventually. Training was slightly faster with learning rate decay, but after one million training games there was no discernible difference in performance. On the other hand, introducing softmax exploration was advantageous because it improved the quality of the training games, resulting in a stronger value function.

- The short-term memory's simulation time is limited, so the increased training efficiency obtained through learning rate decay does increase playing strength. The limited training time also means that ϵ-greedy is a poor simulation policy, since it must compromise severely on either the quality of simulations or its breadth of analysis. The softmax policy performed far better because it naturally explores a variety of lines that are likely in real play. This result is likely to hold in any environment where there are usually several strong actions but where a random action is likely to be weak.

We originally had a more ambitious idea for modifying Dyna-2's exploration policy. The plan was to replace ϵ-greedy with an approach more analogous to UCT, i.e. calculate an uncertainty estimate for the value of each local shape feature, then calculate the uncertainty for each action-value and select the move with the most optimistic upper bound. However, such a policy would not have the same mathematical underpinning as UCB1 [20], since results such as logarithmically bound regret apply only to the n-armed bandit problem. Confidence bounds most likely require an entirely different approach under state abstraction. Sarsa(λ) is not guaranteed to converge [21], so employing a confidence bound that decays to zero as $n \to \infty$ may not work. We still believe that some kind of decaying confidence bound may be viable, but we leave this as an idea for future investigation.

Another modification that could be made, more in keeping with traditional games AI, would be to conduct an alpha-beta search [1, Section 1.4.1] using the combined memories as a heuristic, rather than just selecting the greedy move. In fact, this was already shown to increase the Elo rating of the original *RLGO* from 2030 to 2130 on the Computer Go Server [1, Section 7.4].[3]

Finally, it almost goes without saying that one could try other state representations besides local shape features. For example, the long-term memory could be a neural network, or the features used by the short-term memory could be determined by a meta learning technique. There are many other machine learning techniques that are compatible with Dyna-2.

ACKNOWLEDGMENT

This work is an extension of the PhD thesis of David Silver [1]. We owe David a huge thanks for his initial email correspondence and for making *RLGO*'s source code publically available, without which we could not have conducted any of the experiments described herein.

[2]The tournament policy is 70% greedy / 12% uniform random / 18% Fuego for the first six moves. It then switches to 40% uniform random / 60% Fuego.

[3]There were a number of other improvements that Silver made, such as introducing multithreading and making the program ponder during the opponent's thinking time, which is why the initial rating of 2030 is higher than what we saw in our experiments.

REFERENCES

[1] D. Silver, "Reinforcement Learning and Simulation-Based Search in Computer Go," Ph.D. dissertation, University of Alberta, 2009.

[2] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. Cambridge Univ Press, 1998, vol. 1, no. 1.

[3] Y. Wang and S. Gelly, "Modifications of UCT and sequence-like simulations for Monte-Carlo Go," *CIG*, vol. 7, pp. 175–182, 2007.

[4] A. E. Elo, *The rating of chessplayers, past and present*. Batsford London, 1978, vol. 3.

[5] R. Coulom, "Computing Elo ratings of move patterns in the game of Go," in *Computer games workshop*, 2007.

[6] S. Gelly and D. Silver, "Monte-Carlo tree search and rapid action value estimation in computer Go," *Artificial Intelligence*, vol. 175, no. 11, pp. 1856–1875, 2011.

[7] D. Silver, R. Sutton, and M. Müller, "Reinforcement learning of local shape in the game of Go," in *20th international joint conference on artificial intelligence*, 2007, pp. 1053–1058.

[8] R. Coulom. (2005) Bayeselo. [Online]. Available: http://remi.coulom.free.fr/Bayesian-Elo/

[9] D. Bump, G. Farneback, A. Bayer *et al.* (1999) Gnu Go. [Online]. Available: http://www.gnu.org/software/gnugo

[10] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," in *Machine Learning: ECML 2006*. Springer, 2006, pp. 282–293.

[11] R. Coulom, "Efficient selectivity and backup operators in Monte-Carlo tree search," in *Computers and games*. Springer, 2007, pp. 72–83.

[12] S. Gelly and D. Silver, "Achieving master level play in 9 x 9 computer go." in *AAAI*, vol. 8, 2008, pp. 1537–1540.

[13] M. Enzenberger, M. Muller, B. Arneson, and R. Segal, "Fuego – an open-source framework for board games and go engine based on Monte Carlo tree search," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 2, no. 4, pp. 259–270, 2010.

[14] S. Gelly and D. Silver, "Combining online and offline knowledge in uct," in *Proceedings of the 24th international conference on Machine learning*. ACM, 2007, pp. 273–280.

[15] B. Bruegmann. (1993) Monte-Carlo Go. [Online]. Available: http://www.cgl.ucsf.edu/go/Programs/Gobble.html

[16] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of Monte Carlo tree search methods," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 4, no. 1, pp. 1–43, 2012.

[17] C. J. Maddison, A. Huang, I. Sutskever, and D. Silver, "Move Evaluation in Go Using Deep Convolutional Neural Networks," *arXiv preprint arXiv:1412.6564*, 2014.

[18] M. Müller, "Computer Go," *Artificial Intelligence*, vol. 134, no. 1, pp. 145–179, 2002.

[19] M. I. Jordan, "Why the logistic function? A tutorial discussion on probabilities and neural networks,' 1995.

[20] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2-3, pp. 235–256, 2002.

[21] G. J. Gordon, "Chattering in SARSA(λ) – a CMU learning lab internal report," 1996.